CORE
Metadata, citation and similar papers at core.ac.uk
Provided by Aston Publications Explorer

To appear in the *International Journal of Mathematical Education in Science and Technology*, Vol. 00, No. 00, Month 20XX, 1–8

An elementary algorithm to evaluate trigonometric functions to high precision

[a]* [a] *Anonymous Refereeing*

(*v3.1 released March 2015*)

Evaluation of the cosine function is done via a simple Cordic-like algorithm, together with a package for handling arbitrary-precision arithmetic in the computer program Matlab. Approximations to the cosine function having hundreds of correct decimals are presented with a discussion around errors and implementation.

Keywords: Arbitrary-precision arithmetic, Cordic algorithm, Mulprec package

1. Introduction

A student learning trigonometry might ponder on the question how values of the basic trigonometric functions like $\cos\theta$ are actually evaluated. Typically, a calculator (or computer) is at hands when having to find a particular value. We recall the rather simple geometric idea behind a common method implemented in calculators and computers for evaluating trigonometric functions namely the Cordic algorithm [7]. We shall see that with an additional idea from computer science, termed arbitrary-precision arithmetic (or “bignum” arithmetic), a basic version of Cordic can be used to generate values of the cosine function to hundreds of correct decimal places. To keep the presentation elementary, to be useful for a beginner of trigonometry, we focus solely on the evaluation of cosine; at the end we point to some references on how other elementary functions can be similarly evaluated.

For the outline of the work, in Section 2 derivation of formulas for the Cordic algorithm is given. In Section 3, error analysis is undertaken. Arbitrary-precision arithmetic and motivation of its practical usage are discussed in Section 4 together with a package in Matlab for such arithmetic operations. We also present numerical results in that section and some conclusions. In the Appendix, a code in Matlab with high-precision arithmetic is given for the method discussed of approximating the cosine function.

2. Formulas for evaluation of the cosine function

We shall derive to formulas to be used to evaluate the cosine function. These constitute the basic version of what is known as the Cordic algorithm; for more on the Cordic algorithm and advanced versions the reader can consult [7, 11] and [10, Chapters 8–9]. Our derivation of the basic formulas is shorter than in [10, Chapter 8] since we involve the cosine theorem and the double-angle formula for the cosine function.

*Corresponding author. Email:

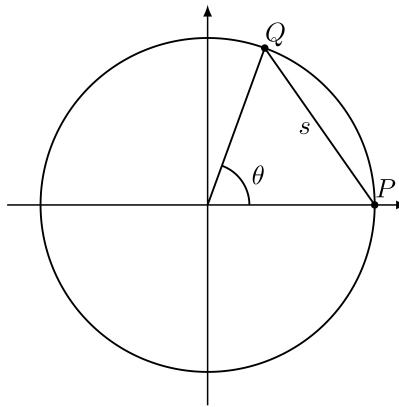

Figure 1. Unit circle with chord of length s corresponding to the angle θ

Consider Fig. 1, where the circle is the unit circle (having radius one) and s is the length of the chord between the points P and Q. Using the cosine theorem, we have

$$s^2 = 2 - 2\cos\theta. \tag{1}$$

This can be written as

$$1 - \frac{s^2}{2} = \cos\theta. \tag{2}$$

Thus, if we know the value of the length s of the chord, the above relationship can be used to find the value of $\cos\theta$.

Taking instead the angle $\theta/2$ and denoting the length of the corresponding chord by h, we similarly obtain

$$1 - \frac{h^2}{2} = \cos\frac{\theta}{2}. \tag{3}$$

The double-angle formula gives

$$\cos\theta = 2\cos^2\frac{\theta}{2} - 1.$$

From this identity using (2) and (3),

$$1 - \frac{s^2}{2} = 2\left(1 - \frac{h^2}{2}\right)^2 - 1.$$

Expanding the bracket in the right-hand side and then solving for s^2, the following is obtained

$$s^2 = h^2(4 - h^2). \tag{4}$$

This means that from the length h of a chord corresponding to the angle $\theta/2$, we can find the length s of a chord corresponding to twice the given angle.

To see the usefulness of this, take for example the angle $\theta = \pi/6$. Half of this angle is $\pi/12$. Now, the length of a chord corresponding to the angle $\pi/12$ is close to the length of the corresponding arc of the unit circle, that is to the value $\pi/12$. Hence, as an approximation, we can take $h = \pi/12$. Using this value in (4), then $s^2 = 0.269458$ rounded to six decimal places. From (2), we find

$$\cos\frac{\pi}{6} \approx \left(1 - \frac{0.269458}{2}\right) = 0.865271.$$

Comparing with the standard value $\sqrt{3}/2 = 0.8660254...$ for $\cos\pi/6$, we obtained a reasonable approximation. To do better, we start with the angle $\pi/24$ and put $h = \pi/24$. Then we calculate the length of a chord corresponding to the angle $\pi/12$ via (4). We re-iterate and call the obtained new length for h and use again (4) to find the length s of a chord corresponding to $\pi/6$. Using s, an improved approximation of $\cos\pi/6$ is found from (2) (the reader trying this will find a slightly better approximation of 0.865838).

Iterations are suitable for a computer to perform. We can with ease do thousands of iterations, however, halving an angle many times it will be rounded to zero on most standard calculators and computers. This can be overcome with what is known as arbitrary-precision arithmetic. Before discussing this, in the next section we briefly investigate the error in the above approximation for $\cos\theta$.

We end this section noting that the angle θ in Fig. 1 is less than $\pi/2$, however, the derivations can be done similarly when the angle is greater than $\pi/2$ although it might be easier to use trigonometric identities to reduce to the case shown in the figure.

3. Some error analysis of the given approximation

The algorithm from the previous section to generate an approximation of $\cos\theta$ is to halve the angle θ a certain number of times k, obtaining a sufficiently small value $\theta/2^k$. Then the length of the chord corresponding to this (small) angle is taken as the length of the corresponding arc of the unit circle, that means $h = \theta/2^k$. Iterate via (4) to obtain the length s of a chord corresponding to the angle θ, and finally apply (2) to find the value of $\cos\theta$.

The error when approximating the length of the chord with the length of the corresponding arc is to be derived. In fact, we derive the error for the difference of the square of the lengths since it is the square being used in (4). We have

$$\theta^2 - h^2 = \theta^2 - (2 - 2\cos\theta) = \theta^2 - \left(2 - 2\left(1 + \frac{\theta^2}{2} + \mathcal{O}(\theta^4)\right)\right) = \mathcal{O}(\theta^4),$$

where we first used the cosine theorem to express h^2 (the square of the chord related to the angle θ) similar to (1) and then applied the Taylor approximation of order 4 of $\cos\theta$. Choosing the angle to $\theta/2^k$ (the corresponding chord is again denoted by h),

$$\frac{\theta^2}{2^{2k}} - h^2 = \mathcal{O}\left(\frac{\theta^4}{2^{4k}}\right). \tag{5}$$

Thus, the error when approximating the square of the length of the chord with the square of the length of the corresponding arc is of order $\frac{\theta^4}{2^{4k}}$. Instead of the correct value h^2, we are according to (5) using a value of the form $h^2 + \mathcal{O}\left(\frac{\theta^4}{2^{4k}}\right)$. To see how this error

propagates when iterating in (4), we find after one step

$$s^2 = h^2(4 - h^2) + 2^2\mathcal{O}\left(\frac{\theta^4}{2^{4k}}\right).$$

Thus, an additional factor 2^2 is introduced. Continuing this k-times and then using (2), the error in the calculation of $\cos\theta$ will be of order $\mathcal{O}\left(\frac{\theta^4}{2^{2k}}\right)$.

Since $2^{-2k} = 10^{-2k\log_{10} 2} \approx 10^{-0.60206k}$, with the choice of $k = 16$ there should be about 10 correct decimals. We can easily do many more iterations on a computer to gain better accuracy with the limitation being the accuracy built into the computer itself. In the next section, it is shown how to work with arbitrary-precision arithmetic in the computer program Matlab to generate hundreds of correct decimal places.

We conclude this section by pointing out that detailed error estimates for a more advanced version of Cordic are presented in [13].

4. Examples of values of cosine using arbitrary-precision in Matlab

Before giving details on arbitrary-precision arithmetic we first briefly indicate that there are practical applications of generating numbers with higher precision than what is usually provided by calculators and mathematical software.

The scale and complexity of computer simulations of real-world phenomena are rapidly increasing, driving the need of more precision. An example where high precision was crucial is the ATLAS experiment at the Large Hadron Collider that verified existence of the Higgs boson. Moreover, in public-key cryptography numbers with thousands of digits are needed. Another example is solving linear systems governed by ill-conditioned matrices (can occur when discretising ill-posed problems) such as the Hilbert matrix (it has an exponentially growing condition number with respect to its size) making the solution to such a system highly sensitive to errors in the data. Further information and motivation for the need of high-precision are presented in [2]. Numerical solutions of some problems generating rapid convergence to at least 10 decimals are given in [12] with one of the examples (problem 2) requiring extended precision. Further techniques for implementing functions with high accuracy is given in [5].

We then turn to giving details of generating high precision approximations and implementation for the outlined method of evaluating the cosine function.

Arbitrary-precision arithmetic is based on the idea of taking a large positive integer B as a number base and representing a number in the usual way like $aB^2 + bB + c$, where a, b and c are the digits, that is integers in the interval $[0, B-1]$. In most of the standard computer programming languages there is some facility to store vectors or arrays of numbers, and such storage is ideal for saving the digits corresponding to the base B. The standard algorithms for addition, subtraction, multiplication and division learned in school can be implemented on such vectors or arrays to get a working arithmetic for elements in the base B. A detailed introduction to arbitrary-precision arithmetic showing how algorithms for standard operations can be realised is given in [6, Chapter 4.3] with history and further references.

In the computer program Matlab, there is a freely available package for arbitrary-precision arithmetic denoted Mulprec; it can be downloaded together with a user manual from http://www.siam.org/books/ot103/. The package is developed in conjunction with the book [3]. The base B chosen in Mulprec is $B = 10^7$. Digits in this base are numbers having at most 7 base 10 digits, stored as elements of a vector (digits in the base B of the multiplication of two numbers can be represented exactly in floating-point in Matlab).

As an example, the assignment

```
x=[-4 3 1415926 5358979 3238462 6433833]
```

means x is a 6 digit mulprec number with, for example, the fourth digit, $x(4)$, being 5358979. The evaluation of x in the standard base 10 is generated as

$$B^{x(1)}\sum_{j=2}^{6}x(j)B^{6-j} = 3B^0 + 1415926B^{-1} + 5358979B^{-2} + 3238462B^{-3} + 6433833B^{-4}$$

$$= 3.1415926535897932384626433833.$$

In the Mulprec package there can be at most 90 base B digits (although as stated in the manual this can in principle be changed to any number of digits). The functions `sum`, `sub`, `mul` and `div` are Mulprec routines based on arbitrary-precision arithmetic for the summation, subtraction, multiplication and division. One can go ahead and use the package without much practise. There are some subtleties like digits being allowed to have varying signs, and for digits starting with a zero the zero itself is left out. Mulprec numbers can always be brought to a standard form with the routine `rnize`. A code for implementing the above procedure for approximating $\cos\theta$ in conjunction with the Mulprec package is given in the Appendix.

From the error analysis of the previous section, the error in the approximation of $\cos\theta$ is of order $10^{-0.60206k}$, where k is the number of iterations, thus with $k = 48$ at least 28 correct decimals is expected. Indeed, running the given code (see the Appendix) with $k = 48$, the result is

$$\cos 2 : -0.41614683654714238699756822950\ldots$$

and comparing with the standard source [1, Table 4.8, p. 175] all 23 decimals given there are exactly matched (checking against sources with more digits, the obtained result differ after the 29th decimal). We are able to match around 600 decimals with $k = 1000$ iterations. Trying larger values of k we run into problems with overflow in Matlab (the code run in about one second for $k = 250$ apart from loading necessary constants and variables needed in Mulprec which takes about the same amount of time in MATLAB R2016b version 9.1 executed on an ordinary workstation having an Intel(R) Core(TM) i3-3217U CPU at 1.80 GHz; no effort has been done to fully optimize the code thus speed can probably be gained by several orders of magnitude by for example vectorising).

For the sake of it, we give the first 50 decimals (obtained with $k = 85$),

$$\cos 2 : -0.41614683654714238699756822950076218976600077107554\ldots$$

We checked the validity of this against the high precision values given by Wolfram|Alpha (https://www.wolframalpha.com). As a side note, $\cos 2$ is irrational [9, Theorem 2.5] (it is also a transcendental number by Theorem 9.11 from the same reference).

In Fig. 2 is the number of decimals obtained with the given code as a function of the number of iterations for the approximation of $\cos 2$. The solid line is the theoretical linear convergence shown in the previous section whilst the stars are the actual number of decimals obtained by the code in the Appendix. Since the obtained decimals closely follow the predicted ones, the code is functioning well and does not unnecessarily increase the approximation error and the same is true for the routines in the Mulprec package (an introduction to errors due to rounding and cancellation is given in [4]).

We have also run the code for other values than $\theta = 2$ and can report the similar accuracy. To calculate some of the standard values like $\cos\frac{\pi}{6}$, a good approximation

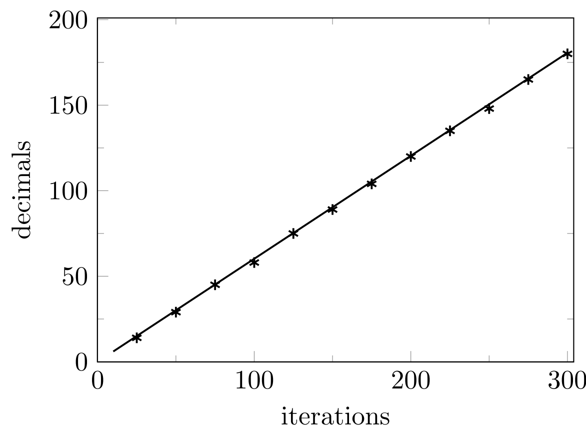

Figure 2. Number of correct decimals as a function of the number of iterations when approximating $\cos 2$. The number of obtained decimals using the code in the Appendix is indicated by a star, $*$.

of π is needed to maintain high accuracy. The given code can be employed to find an approximation to π, as well as to other numbers and functions as will be briefly explained.

Using standard trigonometric identities it is possible to also evaluate $\sin\theta$ and $\tan\theta$ from knowledge of $\cos\theta$. Inverse trigonometric functions and also the exponential and logarithmic functions can be approximated with extensions of the basic Cordic algorithm, see [8, Chapter 6.8 and Chapter 7.2.4] and [10, p. 134 and p. 155] for formulas.

Once $\sin\theta$ has been implemented it can be employed together with the approximation

$$n \sin \frac{\pi}{n} \approx \pi$$

for large values of n to approximate π. Since $\cos\pi = -1$, we can run the basic Cordic algorithm backwards with the starting length of the chord being $s = 2$, then find the length of a chord corresponding to half the starting angle. Iterating we find the value of $\cos\frac{\pi}{2^k}$ expressed in terms of the corresponding chord. Then $\sin\frac{\pi}{2^k}$ is expressed in terms of $\cos\frac{\pi}{2^k}$ and multiplied by 2^k. We tried this rather naive approach obtaining around 200 correct decimals of π for $k = 400$ before getting overflow.

The convergence of the Cordic procedure to evaluate $\cos\theta$ is only linear but the method is still useful since it can be efficiently hard-wired into calculators and computers. This is because halving a number is a fast operation in the binary system due to what is known as bit shifts. Matlab has a version of Cordic built in for trigonometric functions.

To summarise, we shown that using the basic idea of approximating the cosine function via a chord together with a package for high-precision arithmetic in Matlab, hundreds of correct decimals can easily be generated. A software for high-precision arithmetic like the one used can be of benefit not only in practical engineering applications but also in beginning modules on analysis and numerical mathematics to spur discussions around convergence and accuracy. In particular, it should be encouraging to see that simple intuitive iterative schemes can generate arbitrarily high precision approximations.

References

[1] Abramowitz, M. and Stegun, I. A., *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*, National Bureau of Standards Applied Mathematics Series, 55, Washington, D. C. 1964 (Ninth Printing, 1970).

[2] Bailey, D. H. and Borwein, J. M., High-precision arithmetic in mathematical physics, *Mathematics* **3** (2015), 337–367.
[3] Dahlquist, G. and Björck, Å., *Numerical Methods in Scientific Computing*, Vol. I, SIAM, Philadelphia, P. A., 2008.
[4] Einarsson, B., Round-off errors, Encyclopedia of Applied and Computational Mathematics, Ed. B. Engquist, Springer-Verlag, 2015, 1279–1282.
[5] Johansson, F., Efficient implementation of elementary functions in the medium-precision range, in 22nd IEEE Symposium on Computer Arithmetic (ARITH 22), (2015), 83–89.
[6] Knuth, D. E., *The Art of Computer Programming*, Vol. II, Second Ed., Addison-Wesley, Reading, Mass., 1981.
[7] Meher, P. K., Valls, J., Juang, T. B., Sridharan, K., and Maharatna, K., 50 Years of CORDIC: Algorithms, Architectures and Applications, *IEEE Trans. Circuits Syst.*-I **56** (2009), 1893–1906.
[8] Muller, J. M., *Elementary Functions. Algorithms and Implementation*, Birkhäuser, Boston, M. A., 1997.
[9] Niven, I., *Irrational Numbers*, The Carus Mathematical Monographs, No. 11., John Wiley & Sons, New York, N. Y., 1956.
[10] Rising, G. R., *Inside Your Calculator*: *From Simple Programs to Significant Insights*, John Wiley & Sons, New Jersey, 2007.
[11] Schelin, C. W., Calculator function approximation, *Amer. Math. Monthly* **90** (1983), 317–325.
[12] Trefethen, L. N., The $100, 100-digit Challenge, *SIAM News* **35** (2002), No 1: 1 and No 6: 1–3.
[13] Underwood, J. and Edwards, B., How do calculators calculate trigonometric functions?, Educational Resources Information Center (ERIC) document ED461519.

5. Appendix

We give a code using the above outlined algorithm in combination with the Mulprec package in Matlab for approximating $\cos\theta$. The functions `sum`, `sub`, `mul` and `div` are routines in Mulprec based on arbitrary-precision arithmetic for the summation, subtraction, multiplication and division.

```
% Load necessary constants and variables needed in Mulprec
% (package can be found at http://www.siam.org/books/ot103/)
intro

% Choose an angle x (in radians)
x=2;

% The angle x is represented as a mulprec number
x=npr(x);

% Choose the number of steps k of the Cordic algorithm
% k=250 gives about 150 decimals, k=1000 about 600
k=250;

% Base B digits to keep in Mulprec when using elementary
% operations
dec=80;

% The angle x^2/2^(2k) is generated

% Convert 1/2 to a mulprec number
t=npr(1/2);

% Put half=t
half=t;

% t is multiplied by half (k-1)-times to give 1/2^k
for i=1:k-1
    t=mul(half,t,dec);
end

% The number x*t is multiplied by itself to get x^2/2^(2k)
s=mul(mul(x,t,dec),mul(x,t,dec),dec);

% Convert 4 to a mulprec number
four=npr(4);

% The k steps of Cordic is performed.
% Rather than using s^2=h^2(4-h^2) and then
% updating h^2=s^2, it is done in one line
% and s^2 is denoted by s.
for i=1:k
    s=mul(s,sub(four,s),dec);
end

% Formula to use is cos x = 1 - s^2/2, and s^2 is denoted
% by s, rnize(cosx) normalizes the Mulprec number cosx and
% might need to be applied one or more times to the result

cosx=sub(npr(1),div(s,npr(2),dec))
```